Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL

Jonas Tappolet* and Abraham Bernstein

University of Zurich, Switzerland
lastname@ifi.uzh.ch

Abstract. Many applications operate on time-sensitive data. Some of these data are only valid for certain intervals (e.g., job-assignments, versions of software code), others describe temporal events that happened at certain points in time (e.g., a person's birthday). Until recently, the only way to incorporate time into Semantic Web models was as a data type property. Temporal RDF, however, considers time as an additional dimension in data preserving the semantics of time.

In this paper we present a syntax and storage format based on named graphs to express temporal RDF. Given the restriction to preexisting RDF-syntax, our approach can perform any temporal query using standard SPARQL syntax only. For convenience, we introduce a shorthand format called τ-SPARQL for temporal queries and show how τ-SPARQL queries can be translated to standard SPARQL. Additionally, we show that, depending on the underlying data's nature, the temporal RDF approach vastly reduces the number of triples by eliminating redundancies resulting in an increased performance for processing and querying. Last but not least, we introduce a new indexing approach method that can significantly reduce the time needed to execute time point queries (e.g., what happened on January 1st).

1 Introduction

Time, intervals, and versioning are central aspects of many applications. People in organizations, for example, hold many different positions over time, software, comes in many revisions, or services have temporal constraints (such as guaranteed executions times). As a consequence, there exist many data representations of time. Today's approaches, however, code temporal information as additional data inside the data model. Therefore, temporal information is implicit in the data and only difficult to access by programs. Indeed, the semantics of the temporal information needs to be completely coded in the client programs accessing the data. The Semantic Web, however, claims the accessibility of semantics by machines / agents and, therefore, an explicit time representation should support and preserve this central pillar of the Semantic Web.

* Partial support provided by Swiss National Science Foundation award number 200021-112330.

In this paper we propose to use time as an additional semantic dimension of data. Therefore, it needs to be regarded as an element of the meta model instead of being just part of the data model. By applying the foundations established by Gutierrez et al.[1,2], our approach proposes to implement, first, an RDF [3] compatible syntax for temporal data. Second, we provide strategies for efficiently storing temporal RDF data (including an interval based index structure). Last but not least, we introduce a temporal extension to SPARQL [4] called τ-SPARQL and show how τ-SPARQL queries can be automatically translated to standard SPARQL operating on our storage scheme leveraging existing RDF/SPARQL infrastructure.

The paper first discusses the relevant related work. Then, it covers the proposed syntax for temporal RDF and τ-SPARQL. After presenting the mapping of τ-SPARQL to SPARQL it introduces a specialized temporal index structure for temporal RDF, which significantly improves retrieval performance of some types of temporal queries. The paper closes with an empirical evaluation of the introduced techniques and a discussion of the limitations and future work.

2 Related Approaches

The introduction of time into Semantic Web Data structures is not a novel challenge. In this section we will discuss some of these propositions and highlight their strengths and weaknesses. Specifically, we will discuss temporal extensions to OWL and Description Logic, methods for encoding time in the data model, and maintenance of graph versions.

2.1 Temporal Extensions

Different approaches cope with the temporal extension of Semantic Web or Semantic Web related technologies. For example, Artale and Franconi [5] propose to extend Description Logics with temporal features, while Welty and Fikes [6], Kim et al. [7] and the tOWL [8] project aim at introducing temporal entities into OWL. O'Connor et al.[9] investigated the querying of OWL data from the medical domain. In their approach, unlike ours, they use SWRL rules to query the KB. Our goal is to embed time into RDF, i.e., lower Semantic Web layers. We, therefore, base our approach on the foundations introduced by Gutierrez et. al. [1,2]. Gutierrez and colleagues also present a formal sketch for querying temporally enhanced RDF-graphs but did not propose a syntax to be used in, e.g., SPARQL. Even though temporal query languages have been extensively researched in the database community and lead to the TSQL2 query language [10], we were not able to find any analogous extension of the RDF query language SPARQL.

2.2 Time Encoded in the Data Model

A commonly used alternative to encoding time as a semantic element in the representation is to express temporal restrictions and validities, time is modeled as a part of the user's data model. Usually, a datatype property is defined with a range to one of xsd's date datatypes [11]. Pursuing this approach, two different problems arise:

First, we *loose the semantics of time.* Only a human understands that, e.g., the `hasBirthday` property denotes the start of the validity of an instance of the `Human` concept and that the completely different property `manufacturingDate` of the concept `Car` bears semantically comparable information.

Second, *temporal properties can only be attached to concepts or instances.* Whenever a relation needs to be annotated with temporal validity information, work-around solutions such as relationship-reification need to be introduced. As an example, consider the property `isMemberOf`, which is valid in a certain time interval. One work-around would be the reification of the property using a "Membership" blank-node with the properties `start`, `end`, `memberOf`, and `member`. This is cumbersome and leads to an increased query complexity requiring the inclusion of the blank-node in the query's graph pattern.

2.3 Graph Versions and Version Management Systems

Another approach to model time is to maintain multiple versions or temporal snapshots of the graph. Indeed version management systems are very popular to track the evolution of files (such as program source code) and their content. *SemVersion* [12] is such a version management system with a focus on ontology evolution management based on named graphs. However, *SemVersion* uses its own data model that contains the user's ontology and its evolution. Our proposition is not to introduce a new data model, but, storing the elements of the users' data model inside different named graphs according to their temporal context.

The advantage of versioning graphs is that the data model (usually) stays untouched. The major disadvantages are that they (1) limit the temporal semantics to certain time-points (i.e., snap-shots at given points in time) making the use of even slightly more involved temporal semantics, such as interval queries, impossible, (2) do not expose time explicitly but implicitly through snapshots making temporal reasoning complicated, and (3) will force users to access many snapshots if an interval of interest spans many versions forcing them to incur a potentially huge overhead due to redundant information in multiple versions. Our proposed solution will avoid all of these disadvantages.

3 A Temporal Syntax for RDF

In this section we introduce our temporal representations of time. Specifically, we introduce two different representations. The first is for internal use only and facilitates the addressing and indexing of temporal entities (e.g. for a triple store). The second provides the basis for a machine processable exchange format considering the semantics of time.

3.1 Internal Representation

As in most of the approaches dealing with temporal entities, we model time as a 1-dimensional discrete value (i.e. no branching time). More formally, each valid time point is defined as $\tau \in \mathbb{N}_0$. A time interval consists of two time points s (for start) and e (for end) such that $s, e \in \tau | s \leq e$. By allowing s to be equals e, we are able to express time points as intervals. Consequently, we will henceforth only use the interval notation. Note that this integer representation is an abstracted, simplified form of time which looses the exact semantics of time (e.g., whether it is a year number or version number). For most *internal* representations this is sufficient, since we only need to operate on the relative relations of values, i.e., $>$, $=$ and $<$. Furthermore, we allow half-bounded and unbounded intervals. We employ three special intervals that denote time periods which are open in at least one direction: $[0, e]$, $[s, \infty]$, and $[0, \infty]$. Since we restricted the range of valid time points to $\mathbb{N}_0$, 0 denotes the lowest possible value (i.e., the beginning of time). There are no time points before 0. The maximum time value, different from what ∞ might imply, does not lie ahead. It is allocated with the actual time during execution.

3.2 Exposed Representation

As a machine and human legible exchange format an exposed, semantic representation / interpretation is necessary, which may consist of the well-known xsd datatypes. In this approach we use the OWL-Time [13] ontology that defines `Timepoints`, `Intervals` and different date formats. However, we extended OWL-Time with a new date format type called `IntegerTime` to express non-calendaric time representations such as version numbers.

Analogous to the internal and conceptual schema of a DBMS the mapping of the internal to the exposed representation and vice-versa is a storage system dependent strategic decision that impacts the storage and retrieval performance. To map the above mentioned borderline values 0 and ∞ to semantically more expressive external representations, we use the literal values EVER and NOW. Hence, the internal $[0, \infty]$ interval is mapped to its semantic counterpart [EVER,NOW] which refers to a time interval whose elements have always been valid. Any data set without temporal information will be considered to be valid in this "eternal" interval.

3.3 Storage Format and Syntax

As mentioned before, we use named graphs [14] to enable the temporal data representation. We refer to a set of temporally related named graphs as a temporal graph. Each time interval is represented by exactly one named graph, where all triples belonging to this graph share the same validity period. To add an element to a temporal graph, the algorithm shown in Listing 1 is executed.

```
# retrieve validity interval
[start, end] = tripleToAdd.validity
if tripleToAdd containsOneOrMore blankNode
        # generate URI for blank-node
        tripleToAdd = convertBlankToURI(tripleToAdd)
endif
# if no named graph for the interval exists then make one
if !namedGraph[start, end].exists
  create new namedGraph[start, end]
  # add the new named graph to directory in default graph
  namedGraph.defaultGraph += namedGraph[start,end].temporalInfo
endif
namedGraph[start,end] +=  tripleToAdd
```

Listing 1. Pseudo Code for an insert operation into a temporal graph

As the listing shows, we first ascertain the triple's temporal validity. Note that the extraction of the validity is dependant on the way time is represented in a data set. Then, we check, if it contains a blank node. Blank nodes are especially problematic as their validity is restricted to the node's parent graph. In our case, however, where the collection of the interval-representing named graphs logically form the overall graph, it might make sense to have blank-nodes that span multiple intervals and, thus, multiple named graphs. This requirement is akin to the implementation employed by the Intellidimension RDF Gateway's quad approach [14], which deviates from the *W3C* recommendation in that respect. To adhere to the W3C recommendation we simply assign a URI to any blank node circumventing the blank node's spirit but fulfilling our premise of using off-the-shelf tools – in our opinion an acceptable compromise between adhering to the spirit of RDF and practicality. Next, we establish, if a named graph representing the interval already exists. If not, then we generate a new named graph for this interval and add information about it to the named graph directory in the default graph. Finally, we add the triple to the appropriate named graph. Note that we do not allow the same triple to appear in multiple temporal contexts. Therefore, a temporal RDF framework needs to take care that a triple only exists once, but maybe, due to an update of the temporal information travels from one named graph to another.

As apparent from this procedure we use the default graph as a directory or container for the semantic information about the intervals. Since every interval is uniquely identified by the URI of its named graph, vital as well as additional information can be provided in the default graph. Vital information includes the start and end time of an interval in any time format allowed by OWL-Time. Additional information includes relations between intervals such as `after`, `before`, or `overlaps`. Since the additional information about intervals can be inferred from the start and end time of an interval, a temporal reasoning system could entail these relations.

With this approach, we can benefit from well-established, scalable support of named graphs in storage systems without having to implement any adaption to temporal RDF. As mentioned above, temporal validity information could also be provided using reification. However, to express the same amount of information, reification would consume 15 times more triples than named graphs seriously questioning the scalability both in terms of storage space and, more importantly, in terms of retrieval run-time.

4 The τ-SPARQL Query Language

Having established a storage format in Section 3.3, in this section we focus on the retrieval of these triples. It introduces τ-SPARQL, an extended syntax for SPARQL to express temporal queries. We introduce τ-SPARQL by discussing its two major usage formats: time point Queries and temporal queries.

4.1 Time Point Queries

Time point queries aim at retrieving information valid at a specified point in time. This is of special importance whenever a snapshot in the past needs to be retrieved from a dataset. Similar to the `FROM` statement in SPARQL to select a dataset (or graph), we define a `FROM SNAPSHOT` τ expression to select a specific point τ in time. The `FROM SNAPSHOT` τ expression signals the query engine to evaluate the query's graph pattern only on graphs-elements valid at the time point τ, where τ has to be a literal time value (the literal type is depending on the underlying time format). The query in Listing 2, for example, retrieves all `foaf:Person`s that were valid (i.e. alive) in 1995.

```
SELECT ?person FROM SNAPSHOT 1995 WHERE{
        ?person a foaf:Person .
}
```

Listing 2. τ-SPARQL: Time point query

4.2 Temporal Queries

Complementing time point queries that query the graph valid at a given point in time, temporal queries allow the usage of wild card intervals and time points. These wild cards can be used to bind a variable to the validity period of a triple or to express temporal relationships between between intervals. τ-SPARQL allows one form of temporal wildcards, `[?s,?e]`, which binds the literal start and end values. Note that whilst τ-SPARQL explicitly binds the two variables `?s` and `?e` that can be used elsewhere in the query, a third implicit variable `?g` is internally bound to the URI of the named graph that represents the interval defined by *[?s,?e]*. Hence, the expression used in the interval context *[?s,?e]* is bound to the named graph URI while the partial variables *?s* and *?e* are standard SPARQL variables which are bound to the literal start and end value of the interval's validity. Therefore, `?s` and `?e` can occur in any part of the query specified by the SPARQL syntax, e.g. in `FILTER` or `ORDER BY` expressions. In Listing 3, an example query is shown, which retrieves a person's URI and the start of the validity period (i.e. the person's birthday). Since a temporal annotation is based on triple level, the interval variables (or specifications) could be defined before or after each subject, predicate, or object. For the sake of consistency, we define that a temporally annotated triple pattern (i.e., quad pattern) must always start with the temporal variable part.

```
SELECT ?s ?person WHERE{
        [?s,?e] ?person a foaf:Person .
}
```

Listing 3. τ-SPARQL: Selection of validity period

To query for relations between intervals, two interval wildcards can be connected with the properties defined by the OWL-Time ontology. These properties are the well-know Allen Interval Relations [15]. Listing 4 shows a query to retrieve all the `foaf:Persons` that could have known Albert Einstein because their lifespan overlaps with Einstein's. In literature, this kind of patterns are referred to as *temporal join* [10].

```
<PREFIX time: http://www.w3.org/2006/time#>
<PREFIX foaf: http://xmlns.com/foaf/0.1/#>
SELECT  ?s2 ?e2 ?person WHERE{
        [?s1,?e1] ?einstein foaf:name "Albert Einstein" .
        [?s2,?e2] time:intervalOverlaps [?s1,?e1] .
        [?s2,?e2] ?person a foaf:Person .
}
```

Listing 4. τ-SPARQL: Interval relations

5 Mapping τ-SPARQL to Standard SPARQL

In this section we show, that the τ-SPARQL syntax extension can be rewritten without any loss of expressivity to standard SPARQL. Therefore, τ-SPARQL offers an intuitive syntax for the composition of queries (for human consumption), while the rewritten form can be evaluated as standard SPARQL using any query engine supporting named graphs using their built-in standard query optimization and storage formats scalability.

5.1 Mapping Time Point Queries

Rewriting a time point query requires three steps: First, the respective intervals for a time point needs to be determined. Second, the elements belonging to these intervals need to be combined to a new, virtual data set. Finally, the graph pattern needs to be matched against this new data set.

As described above, the information about the validity of an interval is encoded in the default graph of the named graph set. To extract the intervals valid at a given time point, the partial query shown in Listing 5 has to be added to the initial query in the `WHERE` clause.

```
?g time:hasBeginning ?start .
?start time:[inInteger|inXsdDateTime|...] ?s .
FILTER(?s <= <TP> || ?start = NOW) .
?g time:hasEnd ?end .
?end time:[inInteger|inXsdDateTime|...] ?e .
FILTER(?e >= <TP> || ?end = EVER) .
```

Listing 5. Partial query to determine valid intervals at a given time point

The pattern in Listing 5 will bind variable `?g` to all the named graphs containing triples valid in time point <TP>. The expression `[inInteger | inXsdDateTime | ...]` is evaluated according to the literal type of <TP>. If it is a `xsd:Integer`, then the `time:inInteger` property is selected. Analogously, a `xsd:dateTime` will be represented by the `inXsdDateTime` property. Note that both the τ-SPARQL query language and the representation format are open to additional time formats as long as the rewriting algorithm is aware of the acceptable correct formats and their respective mappings.

In a last step of the rewriting process, the initial graph pattern of the query needs to be restricted to the intervals bound in `?g`. SPARQL offers this functionality with the `GRAPH` keyword. This is done by extracting the pattern inside the `WHERE` clause of the query and inserting it enclosed by `GRAPH ?g {<pattern>}`.

5.2 Mapping Temporal Queries

To map the queries that select the validity of a triple pattern and/or relations between validities, the rewriting process is slightly more complex. We define that all classic SPARQL variables `?v` belong to a set C. The intervals `[?s,?e]` to belong to the set of intervals I, and `?s` and `?e` to belong to the set of point variables P. For each $i \in I$ there exists exactly one URI g belonging to the set of URIs of named graphs G and a pair of time points `?s` and `?e`, which mark its start and end. Any pair of time points `?s` and `?e` can be mapped to an interval i, which in turn can be mapped to a URI from G. Consequently, a quad pattern in the form `[?s,?e] <triple pattern>` can be rewritten as `GRAPH ?x{<triple pattern>}` where `?x` is the URI $\in G$ that corresponds to the interval defined by `[?s,?e]`. Note, that a new variable "`?x`" needs to be chosen for every rewrite. Please note that temporal variables are not allowed in the predicate part.

```
<PREFIX time: http://www.w3.org/2006/time#>
<PREFIX foaf: http://xmlns.com/foaf/0.1/#>
SELECT  ?s2 ?e2 ?person WHERE{
        GRAPH g1 { ?einstein foaf:name "Albert Einstein" .}
        ?g2 time:intervalOverlaps ?g1 .
        GRAPH g2 { ?person a foaf:Person }.
        ?g2 time:hasBeginning ?start .
        ?start time:inInteger ?s2 .
        ?g2 time:hasEnd ?end .
        ?end time:inInteger ?e2 .
}
```

Listing 6. Rewritten τ-SPARQL query

It is important that whenever a temporal variable is used outside the interval context (i.e., outside a `GRAPH ?x{<triple pattern>}`) then it needs to be treated like a time point statement. Hence, the partial query shown in Listing 5 needs to be added to the rewritten τ-SPARQL query.

As an example, Listing 6 shows the rewritten standard SPARQL syntax of the query in Listing 4.

6 An Index Structure for Time Intervals

In the previous section we demonstrated how to rewrite τ-SPARQL time point queries to standard SPARQL. However, the performance of time point queries can be raised by providing a dedicated index structure. Temporal index structures have been discussed extensively in the database community. In a recent publication, [16] proposes the tGRIN index structure for temporal RDF w.r.t. the temporal as well as the structural neighborhood of triples. Our requirement, however, is the retrieval of intervals valid in a certain time instant. Therefore, we propose such an index structure that supports the efficient retrieval of intervals valid in certain time instants. This is not a base index of triples, it is rather a meta-index containing named graphs which, by themselves, have indices of their triples.

Specifically, we use a tree-based index structure which stores the validity start and end values in an ordered manner. To achieve a fast retrieval of all valid intervals at a specified time point we would need to store each interval multiple times into this index: in each time point of its validity. E.g., an interval [0,100] would appear 100 times in the index. This is an unpractical approach as it requires a large amount of space. Similar to the mpeg [17] compression algorithm we apply the concept of key frames (more precisely: I frames) to our interval index structure. Similar to a key frame, a key index element stores all the intervals valid at this time point. Between the key indices only the deltas are stored. We refer to this index structure as keyTree index. This index structure is very similar to the Time Index proposed by [18]. The Time index maintains incremental buckets which refer to the deltas between two index elements. However, unlike our keyTree, the Time Index does not allow to select a key index distance, it rather selects the key indexing points whenever an interval starts or ends. This can lead, in datasets with very dense intervals, to oversized indices. Our index offers the parameter *key index distance* which permits a storage system to adapt to different temporal kinds of datasets, e.g., whether the time intervals distribution is dense or sparse. The retrieval strategy of the keyTree index can be summarized as follows:

– if the asked time instant coincides with a key index, return all elements in the key index
– if the asked time instant does not coincide with a key index, go back to the immediately preceding key index. From there, walk forward and apply all the deltas until the asked index element is reached and return the set with the applied deltas.

We will show an evaluation of this index structure in the next section with a comparison to an alternative index.

7 Evaluation

For the evaluation of our approach we are using two different data sources. We used the EvoOnt data [19,20] that consists of software source code information. Specifically, we used information about the source code of the *org.eclipse.compare* plugin of the Eclipse Project. 306 releases created in the period between 2001 and 2007 were available. The second data source is a data set describing Swiss parliamentarians (with information about year of birth and death as well as start and end of service periods) between 1848 and 2008. The latter data has to be seen, for this approach, as a worst case, as almost every triple is defined in its own time interval. Table 1 lists the characteristics of the data sources.

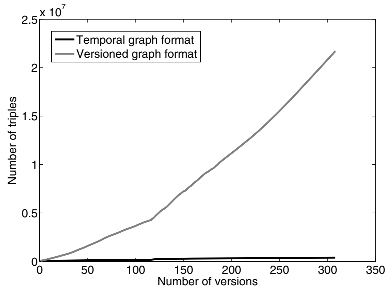
Table 1. Datasets

Data Source	Time Model	Data Period	# Triples	# Intervals
EvoOnt	Versioned graphs	2001 - 2007	22 millions	2505
Parliamentarians	Encoded in data model	1848 - 2008	50'000	38'182

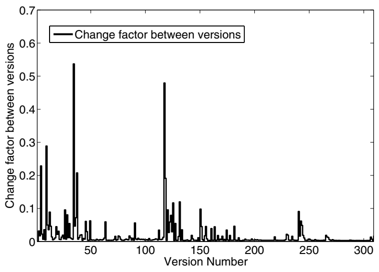
All evaluations in this section were computed on in-memory graphs using the ng4j [21] named graph API. To ensure that we compare different approaches rather than the quality of the implementation of the RDF APIs, we also load the comparison data sets into one single named graph and run queries against the ng4j implementation (which is based on Jena [22]). All the evaluations were executed on an Intel Core2 Duo 1.6 GHz system with 4 GB of RAM running Windows Vista.

7.1 Dataset Conversion

Our first evaluation covers the conversion from the respective time representation format into temporal RDF according to the method presented in this work. First, we converted the EvoOnt data set into a temporal graph. Figure 1(a) shows the vast reduction of triples in the temporal graph approach due to the reduction of redundancies. Having a graph that contains the information about all the 306 versions, the number of triples reduce from 22 millions in the versioned graph approach to 385'000 in the temporal graph. The reason for this huge reduction is depicted in Figure 1(b). The fraction of triples changing between versions of the graph (i.e. between the software versions) is usually very small (far below 5%). As a consequence, 95% of the graph (i.e., the unchanged triples) are stored redundantly.



(a) Growth of number of triples. Gray line: versioned graph, black line: temporal graph.



(b) Change factor between the different versions in the dataset.

Fig. 1. Conversion from graph versions to a temporal graph

7.2 keyTree Index

To evaluate the efficiency of our keyTree index structure introduced in Section 6, we compared our index structure to an in-memory ordered list (Java's TreeList implementation) that contains the intervals' start and end times. Additionally, we added three special sets of intervals containing the borderline values [EVER,x], [x,NOW] and [EVER,NOW] to this baseline implementation.

We evaluated two different scenarios on our datasets. The first is the build up time for the indices with different values for the keyTree index' parameter *key index distance* (i.e. the number of time instants between two key indices). Secondly, we measured the retrieval time for the set of valid intervals in a certain time instant. The following figures show the build and retrieval times for the Parliamentarians (Figure 2) and EvoOnt (Figure 3) datasets. All values were calculated by running each operation 100 times and plotting the average time of these runs.

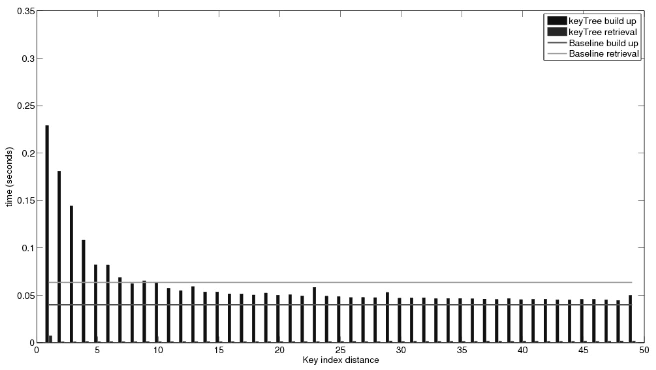


Fig. 2. keyTree index build up and retrieval of the Parliamentarians dataset

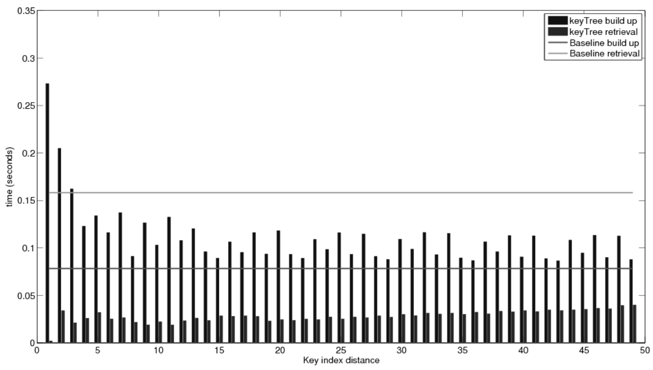


Fig. 3. keyTree index build up and retrieval of the EvoOnt dataset

The figures show, that the keyTree index structure never outperforms the baseline implementation in terms of build time. However, when amortizing the build time over multiple retrievals the use of the more sophisticated keyTree seems to pay off. As a conclusion, the keyTree index is at least three times faster than a "standard" one. The reason for the decreased performance of the baseline is that for each retrieval request the whole index needs to be traversed to check whether there are elements that are valid during the asked time point.

7.3 Timepoint Queries

Next, we run time point queries using both, the Parliamentarians and EvoOnt data sets. Since a time point in the EvoOnt dataset is represented by a snapshot graph, we equate the time to load the graph into memory with the querying time. We also compare the querying time for the temporal graph with the τ-SPARQL triple pattern approach to determine the valid intervals. The evaluated query consisted of retrieving all the triples in a defined time instant (Parliamentarians dataset: 1995, EvoOnt: 5). Additionally we queried the datasets using our keyTree index. Figure 4 shows the execution times of the queries in the different setups and datasets.

Again, the versioned graph approach does not allow to run time point queries, instead, we listed the time needed to load the graph into memory which is the equivalent task needed to select the valid triples in a specific time instant. The results shown in Figure 4 reflect the different nature of the datasets. The Parliamentarian data consists of many different time intervals with relatively few triples in each interval whereas the EvoOnt dataset has less intervals but many more triples per interval. On this note, the keyTree index structure can greatly reduce the query execution time in the EvoOnt dataset because the query engine encounters a vastly reduced number of triples to run the pattern against. The long query execution time for the non-temporal query on the Parliamentarians dataset is mostly due to the complex query pattern using UNIONs to retrieve all elements valid at the desired time point.

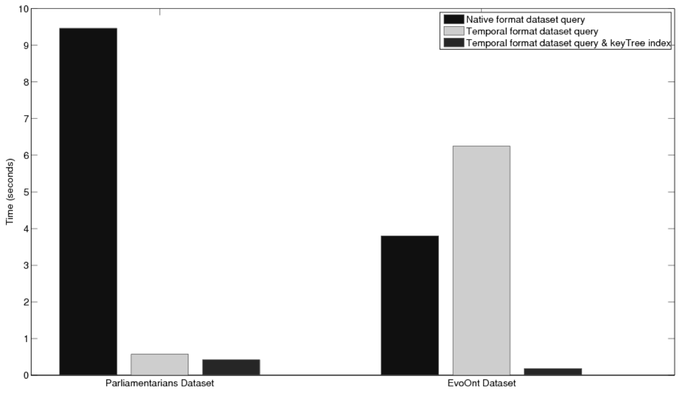


Fig. 4. Execution times of time point queries using different datasets and query strategies

7.4 Temporal Queries

In a final set of evaluations we cover temporal queries. Since such queries (e.g. after, overlaps) are not possible to run against a dataset that uses versioned graphs, we only used the Parliamentarians dataset. The query we evaluated was to retrieve all the parliamentarians complete service periods while "Kurt Furgler" was alive (1924-2008). The τ-SPARQL query uses the `time:intervalDuring` relation, while the classical SPARQL comparison query checks every service period whether it is enclosed by the life time of "Kurt Furgler". Figure 5 shows the execution times of the two queries. Again, it can be seen, that the temporal approach is performing (almost 50%) better because of the reduced complexity of the query pattern. However, the execution time increases whenever all the interval relations are entailed and written into the default graph. For the query, we only entailed the interval relations asked by the query and omitted additional ones. But, as mentioned above, this can be absorbed by using a temporal reasoner that provides the interval relations without materializing them into triples.

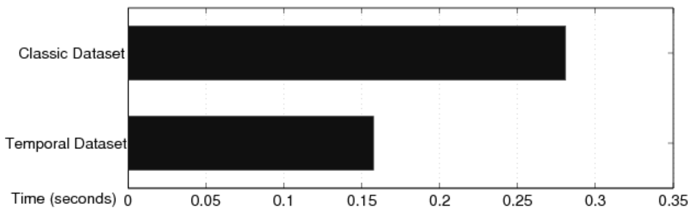


Fig. 5. Execution times of temporal queries using temporal and non-temporal query strategies

8 Conclusion, Limitations and Future Work

We presented an applied temporal RDF using named graphs. In detail, we presented a syntax as storage format which allows the annotation of RDF triples with temporal validity intervals. This approach is as minimally invasive as possible to preexisting storage and querying systems. Furthermore, we introduced T-SPARQL, an extension to express temporal queries. We showed, that τ-SPARQL can be directly mapped to standard SPARQL. Additionally we presented a specific index structure for temporal intervals which improves the retrieval time of time point queries. Finally we evaluated our approach comparing our approach to classical graph versioning and time representation and could show that our temporal RDF and τ-SPARQL approach enables queries that are either impossible (versioned graph) or only achievable with very complex query patterns (time in data model) resulting in increased performance in terms of answering time of queries. We showed cases where our approach outperforms existing ones and circumstances where our approach does not completely outperform existing ones. However, we showed, that a combination with our keyIndex structure can overcome this lack of performance. Moreover, when using temporal RDF, the total number of triples can be largely reduced by removing redundancies which releases resources in each of the storage, reasoning and querying systems. This has a positive effect on disk space usage and response time.

As a limitation we have to mention the lacking availability of temporal reasoners. When materializing the interval relations such as *overlaps*, *meets*, or *before*, the number of temporal relations grows exponentially which makes the materialization in triple form unpractical. Furthermore, our approach performs best on data that has multiple data elements valid within the same interval. If each triple's validity is distinct then one named graph needs to be defined for each triple, which can lead to a decreased performance. However, we believe that many application scenarios exist, where such an extreme interval mapping is not the case.

Our future work will have to concentrate at addressing the current limitations. One possibility is to explore the the entailment of new temporal relations based on our keyTree index. Additionally, we plan to further evaluate our approach with more complex query patterns from different domains of application.

References

1. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing time into RDF. IEEE Transactions on Knowledge and Data Engineering 19(2) (2007)
2. Gutierrez, C., Hurtado, C., Vaisman, A.: Temporal RDF. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 93–107. Springer, Heidelberg (2005)
3. Klyne, G., Carroll, J.J.: Resource description framework (RDF): Concepts and abstract syntax. Technical report, W3C Recommendation (2004)
4. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. Technical report, W3C Recommendation (2008)

5. Artale, A., Franconi, E.: Temporal description logics. In: Handbook of Time and Temporal Reasoning in Artificial Intelligence. Elsevier, Amsterdam (2005)
6. Welty, C.A., Fikes, R.: A reusable ontology for fluents in OWL. In: Proc. of the 4th Int'l. Conference On Formal Ontology In Information Systems (2006)
7. Kim, S.K., Song, M.Y., Kim, C., Jang, H.C.: Temporal ontology language for representing and reasoning interval-based temporal knowledge. In: Proc. of the 3rd Asian Semantic Web Conference (2008)
8. Milea, V., Frasincar, F., Kaymak, U.: Knowledge engineering in a temporal semantic web context. In: The Eighth Int'l. Conference on Web Engineering (2008)
9. O'Connor, M., Shankar, R., Parrish, D., Das, A.: Knowledge-level querying of temporal patterns in clinical research systems. In: Proc. of the 12th World Congress on Health (Medical) Informatics (2007)
10. Snodgrass, R.T. (ed.): The TSQL2 Temporal Query Language. Kluwer, Dordrecht (1995)
11. Campbell, C.E., Eisenberg, A., Melton, J.: Xml schema. Technical report, W3C Recommendation
12. Völkel, M., Enguix, C.F., Kruk, S.R., Zhdanova, A.V., Stevens, R., Sure, Y.: Semversion - versioning RDF and ontologies. Technical report, Institute AIFB, University of Karlsruhe (2005)
13. Hobbs, J.R., Pan, F.: Time ontology in OWL. Technical report, W3C Working Draft (2005)
14. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. Journal of Web Semantics 3(3) (2005)
15. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11) (1983)
16. Pugliese, A., Udrea, O., Subrahmanian, V.S.: Scaling RDF with time. In: Proc. of the 17th Int'l. World Wide Web Conference (2008)
17. Pan, D.: A tutorial on mpeg/audio compression. IEEE MultiMedia 2(2) (1995)
18. Elmasri, R., Wuu, G., Kim, Y.: The time index: An access structure for temporal data. In: Proc. of the 16th Int'l. Conference on Very Large Data Bases (1990)
19. Kiefer, C., Bernstein, A., Tappolet, J.: Analyzing software with iSPARQL. In: Proc. of the 3rd Int'l. Workshop on Semantic Web Enabled Software Engineering. Springer, Heidelberg (2007)
20. Kiefer, C., Bernstein, A., Tappolet, J.: Mining Software Repositories with iSPARQL and a Software Evolution Ontology. In: Proc. of the 2007 Int'l. Workshop on Mining Software Repositories. IEEE Computer Society, Los Alamitos (2007)
21. Bizer, C., Cyganiak, R., Watkins, R.: NG4J - Named Graphs API for Jena (Poster). In: Proc. of the 2nd European Semantic Web Conference (2005)
22. Carroll, J.J., Dickinson, I., Dollin, C., Seaborne, A., Wilkinson, K., Reynolds, D.: Jena: Implementing the semantic web recommendations (2004)
23. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science. Elsevier, Amsterdam (1990)